DyLeCT: Achieving Huge-page-like Translation Performance for Hardware-compressed Memory

Gagandeep Panwar, Muhammad Laghari, Esha Choukse[†], Xun Jian
Virginia Tech [†]*Microsoft Research*
gpanwar@vt.edu, mlaghari@vt.edu, esha.choukse@microsoft.com, xunj@vt.edu

***Abstract*—To expand effective memory capacity, hardware memory compression transparently compresses and packs memory values more densely together in DRAM. This requires introducing a new layer of hardware-managed address translation in the memory controller (MC). However, for large and irregular workloads that already suffer from frequent virtual address translation misses in the TLB, adding an additional layer of address translation can double the translation misses (e.g., by adding a new miss in the MC per TLB miss). While TLB misses can be drastically reduced by using huge pages, no prior work has explored huge-page-like translation reach for hardware memory compression. While compressing and moving an entire huge page worth of data at a time can lead to huge-page-like address translation, moving a huge page worth of data together can consume an exorbitant amount of memory bandwidth.**

This paper explores how to achieve huge-page-like translation performance in this new address translation layer, while keeping compression at the page (instead of huge page) granularity. We propose dynamically shortening the translation entries of hot pages to only a few bits per entry by migrating hot pages to the limited number of DRAM locations whose addresses can be encoded using a few bits; colder pages still use the bigger full-length translations so that colder pages can be placed anywhere in memory to fully utilize all the space in memory. Each short translation is tiny (e.g., 2 bits); as such, a 128KB translation cache filled mostly with short translations can achieve similar (e.g., 2GB) total translation reach as a TLB filled entirely with huge page entries. Evaluations show our idea – Dynamic Length Compressed-Memory Translations (DyLeCT) – improves average performance by 10.25% over the prior art.

I. Introduction

Main memory accounts for a significant portion of operating cost for cloud service providers and hyperscalers. For example, Meta, a prominent hyperscaler, reports that memory accounts for 30% of total hardware infrastructure cost [46].

Hardware memory compression is a promising technique to increase effective memory capacity [6], [7], [17], [27], [33], [37], [43], [52]. It enhances the CPU's memory controller (MC) to transparently compress and pack data more densely in memory; transparently migrating data in turn requires adding a new layer of dynamic address translation beyond the existing virtual-to-physical memory translation. Specifically, the MC manages a linear array of translation entries that we call the compressed-memory translation entries (CTEs). The CTEs collectively form a large translation table that is stored in memory. To avoid fetching a CTE from memory for every memory request, the MC caches CTEs in a dedicated CTE cache, which serves as a similar purpose as the TLB.

This new layer of dynamic address translation, however, can significantly harm the performance of large and irregular workloads. These workloads already suffer from high virtual memory translation overhead (i.e., high number of TLB misses). Adding a new layer of address translation to enable hardware memory compression can effectively double the existing translation misses by adding a new CTE cache miss beyond the old TLB miss.

To optimize virtual memory translation for large and irregular workloads, modern OS and CPU support huge (e.g., 2MB) pages. In our real-system evaluation of large and irregular workloads commonly studied in recent address translation works, using 2MB huge pages provides 1.75x the average program-level speedup over using standard 4KB pages.

No huge-page-like translation, however, exists in the new layer of address translation required by hardware memory compression. Consider a recent prior work [27], which uses a 64-bit translation per 4KB page and a 64KB CTE cache; even after doubling its CTE cache to 128KB, the CTE cache can still only provide a small translation reach of 64MB. Compared to a typical TLB that can provide >2GB reach when the running program uses 2MB huge pages [15], [47], a 64MB translation reach represents a 2GB/64MB=32x lower translation reach.

A naive approach to enable huge-page-like translation reach is to compress and migrate data at 2MB granularity; this allows a single CTE to perform address translation for 2MB worth of data. However, always moving 2MB of data together can consume significant memory bandwidth and severely degrade memory performance.

We note that to fully utilize all the compression-freed spaces, prior works on hardware memory compression deploy fully-associative and non-aligned data placement; but such a highly fluid data placement requires long (e.g., 64-bit) translations. Long translations consume significant space in the CTE cache; this causes the CTE cache to hold relatively few translations and, thus, provide low translation reach.

We observe just like how a mixture of rigid (i.e., harder-to-move) pebbles and fluid water can fill a jar equally well as purely water, using a mixture of fluid data placement and rigid (i.e. lowly-associative and aligned) data placement can also completely fill all space in memory. While a rigid data placement only supports a few (e.g., three) possible DRAM locations per unit of data, this limited choice of DRAM locations only needs short (e.g., 2-bit) translations to encode;

979-8-3503-2658-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ISCA59077.2024.00085

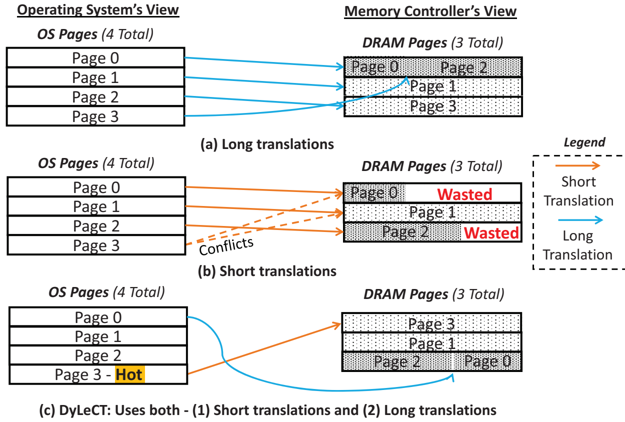


Figure 1: (a) Long translations provide full freedom of data placement to fully utilize all irregularly-sized compression-freed spaces. (b) Short translations restrict data placement and **waste space**. In this example, a short translation can only store an OS page p at the start of DRAM page $d = p\%3$ or DRAM page $d = (p + 1)\%3$ where 3 is the total number of DRAM pages. Dashed line indicates placement conflicts for OS page 3. (c) DyLeCT uses both short translations and long translations. Short translations are more cache friendly and long translations eliminate any wasted space.

a CTE cache can hold many times more short translations (e.g., 64b/2b=32x) than long translations and close the gap in translation reach with a TLB that serves huge pages.

While only some pages can benefit from short translations, we note translation caches generally favor hot pages as they cache the translations of recently-accessed pages. As such, we propose dynamically switching hot pages to short translations and switching cold pages to long translations; this enables the CTE cache to mostly cache short translations and provide high translation reach. We call our idea *Dynamic Length Compressed-Memory Translations* (DyLeCT[1]). DyLeCT dynamically migrates hot pages to the limited DRAM locations whose addresses short translations can encode, by displacing cold pages to DRAM locations whose addresses only long translations can encode (see Figure 1c). While designing DyLeCT, we address a key challenge of how to minimize data movement overheads and also redesign the CTE table and cache organizations to tailor to dynamic-length translations.

Our evaluation shows that DyLeCT improve the performance of translation-intensive workloads by 10.25% over the prior art. Table 1 compares DyLeCT to prior works.

We make the following contributions in this paper:

1) We explore how to enable huge-page-like translation performance in the new address translation layer introduced by hardware-compressed memory.
2) We propose Dynamic Length Compressed-Memory Translations (DyLeCT) to dynamically switch between short and long translations on a per-page basis to achieve the best of both worlds – high translation hit rate and high memory density/capacity.
3) We address the unique design challenges of how to effectively use both types of translations together.
4) Our evaluations show DyLeCT improves average performance by 10.25% over a recent prior work –TMCC– without sacrificing memory compression ratio. DyLeCT is also a simple design that only modifies a single hardware component – the memory controller.

[1]It is pronounced as *Dialect*, as the two types of translations are different ways or dialects of communicating the same DRAM address.

Prior Work	Comp. Ratio	Perf. Improvement	Modifications vs. Current Systems
RMC [7]	1.30x	N/A	MC
LCP [33]	1.69x	+6% vs RMC	MC, TLBs
Compresso [6]	1.85x	+6% vs LCP	MC
TMCC [27]	3.40x	+14% vs Compresso @ 1.85x Comp. Ratio.	MC, L2$
This Work	3.40x	**+10.25% vs TMCC** (under huge pages)	**MC**

Table 1: Contrasting DyLeCT with prior works.

II. Related Works on Hardware Memory Compression

Many prior works propose enhancing the memory controller (MC) to dynamically compress data in DRAM; they propose adding decompressor and compressor hardware to the MC to read and write compressed data in an OS-transparent manner. These prior works broadly fall under two categories.

One category uses compression to save memory bandwidth [13], [16], [17], [48], [49]. They compress data and store them in or near their original locations and, therefore, leave many unused compression-freed spaces scattered across DRAM; as such, they do not improve effective memory density/capacity.

The second category uses compression to increase the effective memory capacity exposed to the OS [6], [7], [27], [33], [37], [43], [52]. These works migrate compressed data closely together to improve effective memory density/capacity. As such, they can enable the OS to use several times (e.g., 4x) more OS physical memory than the amount of DRAM in the system. This can help avoid swapping out data when memory pressure is high[2]. Minimizing swapping out in turn also helps to preserve huge pages; before swapping out, today's OS first breaks down huge pages into standard pages because directly swapping huge pages can be prohibitively expensive.

This paper focuses on using hardware memory compression to improve effective memory capacity.

A. Background on Memory Address Translation for Hardware Memory Compression

Hardware memory compression transparently migrates compressed data to pack it densely in DRAM. Prior works track this data movement by adding a new layer of hardware-managed translation that consists of a linear array of *compressed-memory translation entries (CTEs)*. Instead of storing a tuple of *col:row:bank:channel IDs*, each CTE records

[2]Even when memory pressure is low, freeing up memory via hardware memory compression could still be useful as free memory can be used to opportunistically boost memory performance [28], [32], [51].

the location of data in DRAM as a *scalar address* called the *machine-physical address* [6], [31]. The machine-physical address is then translated into *col:row:bank:channel IDs* using the same static address mapping function that today's systems use to calculate the final DRAM location given a conventional physical memory address [34].

The CTEs are stored in a statically reserved memory region called the CTE table. The CTE table contains CTEs for the entire OS-visible memory. Each chunk of OS-visible memory has its own dedicated CTE. We call the size of this chunk of OS-visible memory the *translation reach* of the CTE. The CTE table is a simple flat table; the nth entry in the CTE table corresponds to the nth regular sized chunk of OS-visible memory. We refer to each 4KB range of OS-visible memory *an OS page* regardless of whether it is being used standalone as a standard page or is currently a part of a huge page.

B. Prior Works on Optimizing the New Address Translation

On a last-level cache (LLC) miss, accessing the CTE to determine the location of the requested data increases the critical path of the LLC miss. To speed up the translation, all prior works add a small cache to the MC to cache CTEs (see Figure 2).

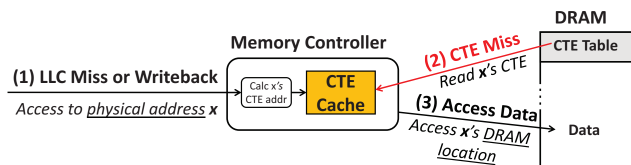


Figure 2: In hardware memory compression, CTEs form an additional address translation layer beyond the conventional virtual memory translation layer. Every memory access goes through one extra layer of indirection that translates physical address to a DRAM location with the help of a CTE.

Large applications that can benefit the most from compression, however, suffer from high CTE cache miss rates, especially when they also have irregular access patterns. These large and irregular applications are also known to suffer frequent TLB misses. Further exacerbating the address translation problem, many prior works [6], [7], [33], [52] compress and move individual memory blocks, which requires fine-grained tracking and address translation. In general, finer translation granularity increases translation miss rate. For example, Compresso [6], which compresses individual memory blocks, uses a 64B CTE per 4KB page, instead of 8B like a page table entry (PTE); bigger CTEs are more likely to overwhelm the small CTE cache in the MC.

To keep frequently-accessed CTEs small, TMCC keeps hot pages uncompressed so that they can use coarse-grained page-level translation. TMCC divides memory into a two-level exclusive hierarchy. Memory Level 1 (ML1) stores hot pages in uncompressed form to keep the CTEs small; Memory Level 2 (ML2) stores cold pages in compressed form to increase effective memory capacity. Memory Level 2 compresses data at 4KB page, instead of 64B block, granularity so that the CTE entry managing a page can serve the page regardless of whether the page is uncompressed or compressed; page-granularity compression also enables higher (e.g., 2x) compression ratio. For every LLC miss or writeback request that accesses a page in ML2, TMCC decompresses the page and stores it in uncompressed format as part of ML1; this page promotion is called a ***page expansion***.

While increasing translation granularity to the page-level reduces CTE cache miss rate, the miss rate is still high for large and irregular workloads. As such, TMCC also proposes to compress page table blocks (PTBs)[3] in the cache hierarchy to embed CTEs within PTBs. Embedding CTEs within PTBs enables a normal page walker access to also obtain from the accessed PTB the CTE that the data access after the walk will need; this CTE obtained from the accessed PTB can then be forwarded to the MC along with the cache miss request for the data. The MC can then use the CTE to satisfy the miss request, without incurring any translation latency overhead.

Because TMCC is the most recent work on optimizing address translation for hardware memory compression, we build upon aspects of TMCC in this paper. Below, we describe aspects of TMCC that we will also preserve in this work.

CTE Table: TMCC stores CTEs in an unified CTE table where each CTE translates for a 4KB OS page. This is a flat table, where the nth CTE corresponds to the nth OS page. As each CTE in TMCC is 8B long, a 64B memory block of the CTE table stores 8 CTEs. At the time of a CTE cache miss, the MC fetches one 64B memory block from the CTE table; to be concise, we call this block a *CTE block*.

Managing Free Space: To track free memory, TMCC maintains a linked list of free 4KB DRAM pages called the Free List. TMCC also has many other free lists that track irregular-sized free spaces of <4KB; each list tracks free spaces of a specific size (e.g., 1.5KB).

Managing In-use Space: Memory that is not part of any of the free lists is in-use. When a memory request accesses the data in an ML2 page (i.e., a compressed page), TMCC expands the page into a free 4KB DRAM page from the Free List (see Figure 7a). When compressing an uncompressed page in ML1, TMCC stores the newly compressed page into a tightly-fitting irregular free space tracked by one of the free lists.

Compressing Least-Recently-Used ML1 Page: To select a victim for compression, TMCC maintains a Recency List tracking all uncompressed pages. Once every 100 memory requests, TMCC updates the list's head to point to this most-recently accessed page; this naturally causes less-recently accessed pages to drop down in the list so that the list's tail points to the least-recently accessed page.

Demand-adaptive Compression: TMCC adaptively compresses data in response to memory pressure to maintain 16MB of free DRAM pages in the Free List. When free memory is <16MB, the MC compresses pages asynchronously (i.e., in the background) by repeatedly compressing pages from the tail of Recency List and then using the freed-up space to replenishes the Free List.

[3] A page table block is a 64B memory block containing 8 PTEs.

III. Motivation

Contemporary workloads such as databases, graph analytics, and machine learning have large memory sizes of many tens of gigabytes [8], [11], [12], [23], [45]. Many large workloads also have irregular memory access patterns that lead to high TLB miss rates. Owing to the popularity of these workloads, many prior works actively explore how to minimize virtual memory translation overheads for these workloads [1], [23], [29], [30], [35], [41].

To handle irregular workloads, current systems support 2MB huge pages [9], [19], [26], [53], [54] and deploy them in large (e.g., Google, Amazon, Meta) datacenters [14], [25], [38], [44]. Recent Linux distributions also turn on transparent huge pages by default to help minimize TLB miss rates for irregular applications [22], [39]. On a real x86 machine, we find 2MB huge pages can provide 1.75x average speedup for large and irregular applications (see Figure 3).

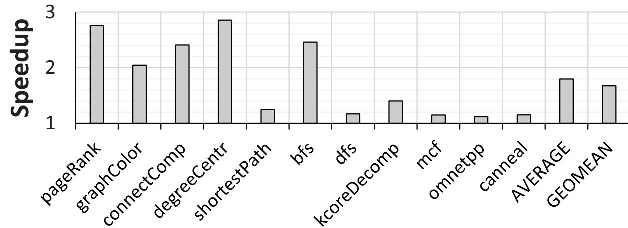


Figure 3: Speedup when running large irregular workloads under 2MB huge pages vs 4KB standard pages on a real system with Intel W-3175X CPU. The high speedup of 2MB huge pages comes from both improved address translation performance and faster page allocation. The evaluated benchmarks are used by recent prior works on hardware memory compression and by prior works on address translation for current systems without memory compression [23], [27], [30]; they come from GraphBIG [24], SPEC CPU2017 [42], and PARSEC 3.0 [3] benchmark suites.

A. No Prior Work on Hardware Memory Compression Effectively Optimizes Address Translations under Huge Pages

TMCC's primary translation optimization – embedding truncated CTEs within PTBs (see Section II-B) – only works during page walks. But page walks are rare under huge pages; 2MB huge pages reduce TLB miss rates by 20x, on average, in our real-system experiments in Figure 3. Furthermore, each 64B PTB for 2MB huge pages has 8 PTEs and, therefore, covers eight 2MB pages or 16MB of memory; the many CTEs of the many constituent 4KB pages within the 16MB of memory are too numerous to fit in a single 64B PTB.

Without the above optimization, TMCC's only translation optimization is to reduce the size of each CTE to 8B, down from 64B or more under earlier prior works; however, 8B per 4KB page still incurs high translation miss rate, just like how standard pages incur high TLB miss rates in today's systems for large and irregular workloads. To quantify performance under huge pages, we implement TMCC in Gem5 (see methodology in Section V) and run the workloads under 2MB huge pages. Compared to a bigger conventional memory system without any compression, TMCC suffers 14% and 18% average performance loss at low and high compression, respectively (see Figure 4).

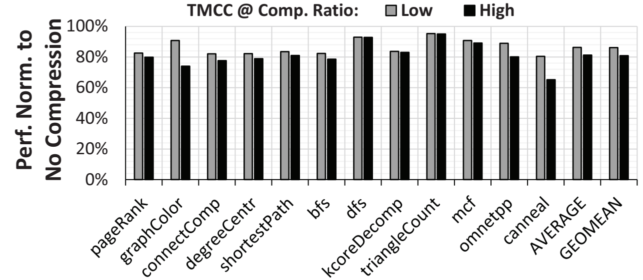


Figure 4: TMCC's performance normalized to a bigger memory with no compression. Since TMCC compresses memory adaptively according to the current memory demand, it can be evaluated under different degrees of compression. The 'Low Compression' and 'High Compression' settings are taken from the TMCC paper [27]. Table 2 shows each benchmark's original memory footprint and the DRAM size used to evaluate them under each setting. The CTE cache size is 128KB.

Benchmark	Memory Footprint	DRAM @ Low Comp.	DRAM @ High Comp.
GraphBig Suite	106GB	81.5GB	35GB
mcf	15GB	13.7GB	6GB
omnetpp	1GB	0.63GB	0.4GB
canneal	1.1GB	0.96GB	0.73GB

Table 2: Evaluated benchmarks and DRAM size for simulations.

Much of the performance loss is due to the poor translation cache hit rate. A 128KB cache only provides a small translation reach of (128KB/8B)*4KB=64MB. Such a small translation reach leads to a high average miss rate of 28% for GraphBig (see Figure 5). As sensitivity analysis, we also sweep other CTE cache sizes – 64KB, 256KB, and 512KB. Octupling the CTE cache size from 64KB to 512KB only reduces CTE cache miss rate from 34% to 24%. Making the CTE cache even bigger would make it slower to access and, in turn, slow down all read requests from the LLC, including those that *hit* in the CTE cache.

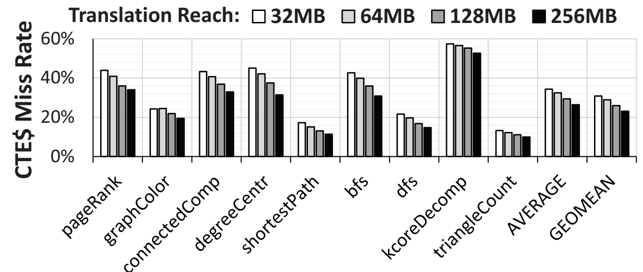


Figure 5: CTE cache miss rate for large workloads under 2MB OS huge pages. Octupling the translation reach by octupling the CTE$ size only reduces the average miss rate by 10%.

B. Hardware-managed Large Pages can be Harmful

A plausible solution to increase translation reach is to increase compression granularity from 4KB to 2MB, so that

each CTE can translate for 2MB of data. This increases each CTE to the same translation reach as a 2MB huge page PTE. However, compressing and decompressing 2MB each time can incur an exorbitant bandwidth overhead. Irregular workloads have a skew in accesses within a huge page [20] (i.e., only some of the 4KB pages within each huge page are frequently accessed); as such, always moving everything belonging to a 2MB page together as a single unit via a single CTE can waste significant bandwidth. Furthermore, 2MB compression also increases decompression latency to 100μs (i.e., 512*decompression latency of 4KB DEFLATE ASIC decompressor [27] = 512*280ns = 143.36μs).

An alternate solution is to use a compression granularity less than 2MB, but still coarser than 4KB (e.g., 64KB). This also increases the translation reach of CTEs. However, the bandwidth overhead in this case is an order of magnitude lower than 2MB compression granularity.

To quantify the performance of coarse-granularity compression, we also evaluate TMCC at three coarse granularities: 16KB, 64KB, and 128KB. Under low compression, 16KB, 64KB, and 128KB compression reduce the average slowdown from the earlier 14% down to 9.5%, 7%, and 6%, respectively (see Figure 6). This is because having a coarser compression granularity increases the translation reach of the CTE cache; for example, 64KB compression granularity improves the translation reach of CTE cache by 64KB/4KB=16x.

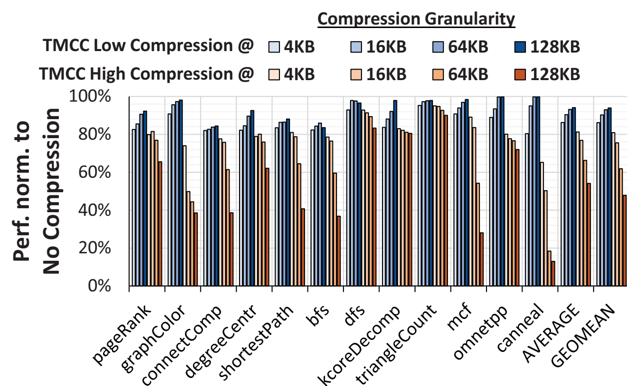


Figure 6: TMCC at 4KB, 16KB, 64KB and 128KB compression granularities. Performance normalized to a system with no compression.

Under high compression, however, performance decreases as compression granularity increases; the slowdown are 18%, 23%, 34%, and 46% when compressing at 4KB, 16KB, 64KB and 128KB, respectively (see Figure 6). Under high compression, decompression and compression become more frequent; the high bandwidth overhead of decompressing and compressing at coarse granularity outweighs the benefit of increased translation reach. If the compression granularity were 2MB, the slowdown would be even worse. As such, the severe performance degradation at high compression ratio rules out coarser compression as a practical design point.

IV. Dynamic Length Compressed-Memory Translations (DyLeCT)

In this paper, we explore how to achieve a huge-page-like translation performance in the new layer of address translation required by hardware memory compression, without increasing the granularity of compression (i.e., keep it at 4KB).

We note that all prior works on hardware memory compression use fully-associative and non-aligned data placement; such fluid data placement can store data in arbitrary compression-freed spaces and, therefore, help maximize the logical density in memory. In comparison, CPU caches use a set-associative data placement, which stores a given memory block or page at few possible locations; while more rigid, set-associative placement requires fewer bits per translation (e.g., 2 bits if 3 ways) and, thus, can greatly shrink the size of each CTE (e.g., by 8B/2b=32x) and greatly increase translation reach (e.g., by 32x, as a cache can fit 32x as many CTEs if each CTE is 32x smaller). However, a rigid placement cannot utilize most of the compressed-freed spaces and wastes memory, which defeats the purpose of hardware memory compression. As such, no prior works use set-associative data placement.

To achieve the best of both worlds, we observe just like how a mixture of fluid water and a mixture of rigid (i.e., harder-to-move pebbles) can fill a jar equally well as purely water, using a mixture of fluid data placement and rigid data placement can also completely utilize all space in memory. As such, we propose using a combination of *long translations*, which support fluid data placement to preserve the same effective memory capacity as prior works, and *short translations*, which support set-associative data placement to improve the translation reach of the CTE cache. Furthermore, since only a portion of the data in memory can use long translations, we propose dynamically and transparently switching translation length for individual OS-visible pages depending on how frequently they are accessed. We call our proposal *Dynamic Length Compressed-Memory Translations (DyLeCT)*.

A. Challenges of Dynamically Switching the Length of CTEs

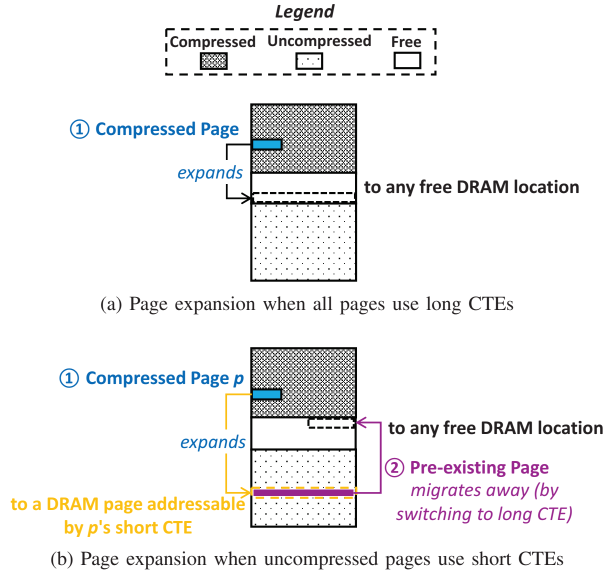
Noting that the recent prior work on optimizing address translation for hardware memory compression (i.e., TMCC) keeps hot pages uncompressed, a basic DyLeCT design is to use short translations on uncompressed pages (i.e., the Memory Level 1 under TMCC, see Section II-B), as they are hot; meanwhile, use long translations on compressed pages (i.e., Memory Level 2, see Section II-B) to utilize all compression-freed spaces in memory.

While dynamically switching CTE lengths may seem rather simple, it faces several design challenges.

1) Bandwidth Challenge: Dynamically switching between CTE length can incur a costly bandwidth overhead. When using long CTEs for all pages (i.e., both ML1 and ML2), after a compressed (i.e., ML2) page becomes hot again due to an access, the page can expand directly to ***any*** free DRAM page (see Figure 7a) that is being tracked by the Free List. When each uncompressed page uses the short CTE, however, *every one* of the few possible locations that the page's short CTE

can address/encode is very likely *already in use*, especially in a highly-occupied memory system that needs compression; as such, expanding an ML2 page to ML1 would require first moving one of the pages currently occupying one of these DRAM pages to a free DRAM location somewhere else and then expand the accessed page into the freed-up DRAM page (see Figure 7b).

Having to move two pages per page expansion can double the bandwidth overhead of page expansions over always using long CTEs. *How to mitigate this costly 2x bandwidth overhead for page expansions is a challenge.*



(a) Page expansion when all pages use long CTEs

(b) Page expansion when uncompressed pages use short CTEs

Figure 7: (a) Page expansion when all pages use long CTEs. The page can directly expand to a free DRAM page tracked by the Free List. (b) Page expansion when ***all compressed pages use long CTEs***, but ***all uncompressed pages use short CTEs***.

2) Cacheability Challenge: Prior translation table designs both in hardware memory compression and conventional systems adopt a unified design that uses a single CTE table for both uncompressed and compressed pages and uses a single multi-level table for both standard and huge pages. Reusing such a unified table design for DyLeCT would still yield poor CTE cache hit rate. This is because under a unified CTE table design, many of the bits in a table entry are unused when the entry stores a short CTE; caching a CTE block containing many unused bits wastes precious space in the CTE cache and, thus, reduces the CTE cache hit rate.

Figure 8 shows a 64B block for the unified CTE table where each of the 8B entries in the block serves a 4KB OS-visible page. Each 8B entry holds either a long translation or a short translation. Using 8B to record a short translation wastes significant space. If the unified block is cached naively (i.e., cache the block with all of its unused bits), the cache will store no more translations than using long entries for all pages.

To reduce waste in the CTE cache, a potential solution is to have separate caches for short and long CTEs. After accessing

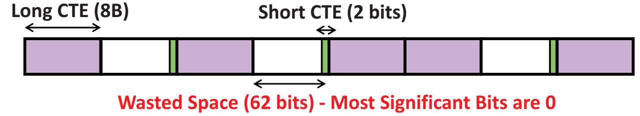


Figure 8: An example of a 64B block under the unified CTE table organization. Each block contains 8 8B entries; when an entry records a short CTE the most significant 62 bits in the entry are wasted.

a 64B unified CTE block, the memory controller (i) inserts an 8B long CTE into the dedicated long CTE cache if the CTE is used and (ii) gathers all short CTEs closer together before inserting them into the dedicated short CTE cache. However, having two distinct caches for short and long CTEs is inefficient as the long CTEs are used less often.

Furthermore, the short CTE cache can still waste significant space. One option for designing the short CTE cache is to gather all short CTEs that are part of the fetched unified block into a small cacheline in the short CTE cache; for 2-bit short CTEs, this means gathering eight 2-bit short CTEs into a 2B cacheline. The problem is that each tag in such a short CTE cache can be much bigger than the data in each cacheline (e.g., a 4B tag versus 2B cacheline data), which can waste significant (e.g., 66% of) the cache area (see Figure 9 ‘Option A’). Another option is to organize the short CTE cache as a sector cache [2], [40] that uses 64B cachelines instead of 2B cachelines to amortize the tag overhead. The downside of this approach is requiring a long time to warm up each 64B cacheline and, therefore, wasting most of the bits in the common case (see Figure 9 ‘Option B’).

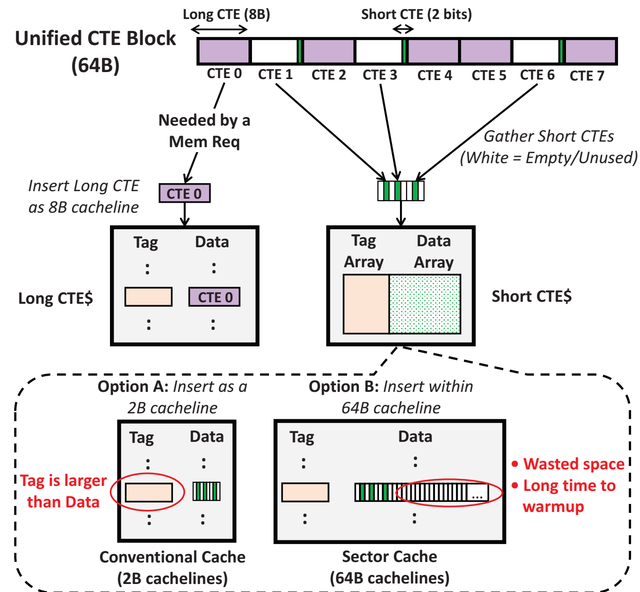


Figure 9: Basic approaches to address the cacheability challenge; they still waste significant space in CTE caches.

3) Quantifying the Two Challenges: Using Gem5 (see methodology in Section V), we simulate the basic design that

dynamically switches between long and short CTEs within a unified CTE table. It switches between 2-bit short CTEs for uncompressed pages and 8B long CTEs for compressed pages. It models movement of two pages per expansion (see Section IV-A1). It has two 64KB CTE caches that store short and long CTEs, respectively. When the memory controller reads a CTE block, it gathers up to 8 short CTEs from the block into a single 2B cacheline in short CTE cache and inserts the block's long CTE(s) into 8B cacheline(s) in long CTE cache.

We evaluate the benchmarks in Section V at high compression setting (see Table 2) and find the average CTE cache hit rate is 76% – only marginally better than the 67% hit rate under TMCC; this improvement is small due to inefficient use of space in the CTE caches. Furthermore, the bandwidth overhead due to double page movement per expansion degrades performance and masks any potential performance benefit from slightly improving CTE cache hit rate. Consequently, instead of improving performance, this naive design actually reduces performance by 5% on average.

B. Addressing the Bandwidth Challenge via a Three-level Memory Hierarchy

To address the bandwidth overhead challenge of double page movement per page expansion (see Section II-B), DyLeCT uses both short and long CTEs for uncompressed pages. When first expanding a compressed page to uncompressed form, DyLeCT uses a long CTE to store the page in any free DRAM page that is currently being tracked by the Free List. DyLeCT only selectively switches the hottest uncompressed pages to using short CTEs. Dynamically switching between short and long CTEs for uncompressed pages essentially extends the two-level memory hierarchy into a three-level exclusive hierarchy, where the hottest uncompressed pages form Memory Level 0, while less hot uncompressed pages form Memory Level 1 (see Figure 10).

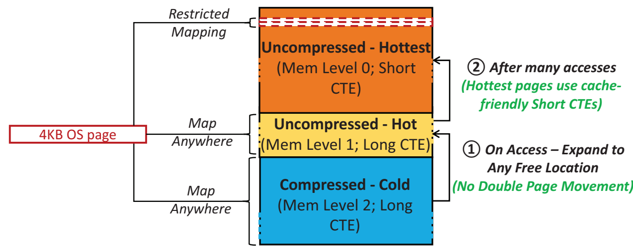


Figure 10: Three-level exclusive memory hierarchy in DyLeCT. Memory Level 0 stores hottest OS pages in uncompressed form. DyLeCT accesses data in this level through cache-friendly short CTEs. Memory Level 1 also stores OS pages in uncompressed form but uses long CTEs. Memory Level 2 stores compressed data and also uses long CTEs. For clarity, the figure shows the different memory levels occupying contiguous memory; however, ***they can be non-contiguous and arbitrarily interleaved.***

Memory Level 0 (ML0): ML0 stores uncompressed pages and addresses them using short CTEs. A short CTE of an OS page p can only place p among a small set of possible DRAM pages (e.g., a 2-bit short CTE of p can only place p in one out of 3 possible DRAM pages); we refer to this set of DRAM pages as p's *DRAM page group*. The DRAM pages within a DRAM page group are adjacent to each other. Two distinct OS pages can either share the DRAM page group or use distinct DRAM page groups that do not overlap.

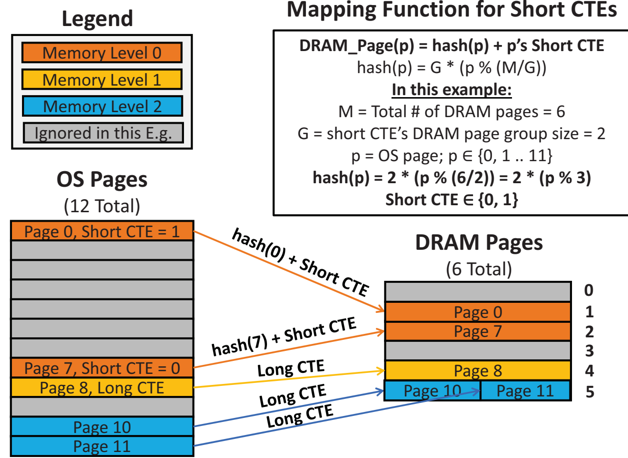


Figure 11: **Short CTE:** Our mapping function for short CTEs. In the small example memory system, OS page 7 in ML0 is stored at $DRAM_page(7) = hash(7) + ShortCTE = 2 + 0 = 2$. **Long CTE:** The long CTEs are not accompanied by any calculations; each long CTE *directly* records the current machine-physical address of an ML1 or ML2 page.

DyLeCT uses a static hashing function to identify the first DRAM page in the DRAM page group of an OS page p. The hash function $hash(p)$ takes as input p's page ID, the total number of DRAM pages in the system (M), and the number of DRAM pages per DRAM page group (G). The full hash function is given in Figure 11 *hash(p)*; the multiplication by G ensures two adjacent OS pages map to two distinct DRAM page groups.

The short CTE of page *p* then specifies which one out of the G DRAM pages in the DRAM group is currently storing p. Therefore, the complete mapping function used by short CTEs is $DRAM_Page(p) = hash(p) + p's\ Short\ CTE$. Figure 11 illustrates how to use short CTEs to locate ML0 pages in an example system with 12 OS pages and 6 DRAM pages.

Memory Level 0 is dynamic in size; it may scale up to the *entire memory system* when everything is uncompressed (e.g., when the memory pressure is low). This is because the output range of the hashing function for short CTEs is the entire DRAM; $hash(p) = G*(p\%(M/G))$ approximately simplifies to $p\%M$, where M is the entire DRAM size. As such, any DRAM page can be part of ML0; in other words, any DRAM page can store an uncompressed page that is currently using a short CTE.

Memory Level 1 (ML1): ML1 is the next level of memory that also stores uncompressed pages. However, unlike ML0, ML1 pages uses long CTEs so that they can be stored anywhere in memory. Long CTEs are 8B each so that they can encode arbitrary DRAM addresses.

Memory Level 2 (ML2): ML2 stores compressed pages and uses long CTEs to address them.

ML2→ML1 Promotion (① in Figure 10): A potential promotion policy for DyLeCT's three-level hierarchy is the conventional promotion policy used in CPU caches: expand a page from ML2 directly to ML0 similar to how the CPU promotes a cacheline from L3$ directly to L1$. However, such a policy continues to incur double page movement per page expansions (see Section IV-A1). Heuristically, a recently expanded page is unlikely to be very hot (e.g., it was compressed initially because it was cold) and may receive very few accesses before it is compressed again. We confirm this for all irregular workloads in Section III and find that on average across all benchmarks, a decompressed page receives 16 accesses before it is compressed.

DyLeCT adopts a gradual promotion policy that first expands a compressed page in ML2 to ML1, and then selectively promotes ML1 pages to ML0 (see ML1→ML0 Promotion). *The expansion of compressed pages to ML1 uses free pages addressable by long CTEs to avoid double page movement.*

ML1→ML0 Promotion (Long CTE→Short CTE Switch; ② in Figure 10): DyLeCT selectively promotes the most frequently accessed pages from ML1 to ML0. We note how to select the hottest pages to place into a limited set of page-sized locations is quite similar to prior works on DRAM caching at the page granularity. As such, we adapt the promotion algorithm from a prior work on page-level DRAM caching (specifically, 'Algorithm 1' from [50], with 5% sampling rate) by maintaining a probabilistic access counter for every OS page. The hot ML1 pages to promote are identified as the ones with access counts that are higher by a threshold than other ML1 pages that map to the same DRAM page group.

When DyLeCT promotes a hot ML1 page p, some of the DRAM pages in p's DRAM page group may contain ML1 or ML2 pages; DyLeCT uses the long CTEs of these ML1 or ML2 pages to migrate them elsewhere to free up a DRAM page to store p. Page p is now in ML0.

ML0→ML1/ML2 Demotion (Short CTE→Long CTE Switch): When DyLeCT promotes a page p, if all of the DRAM pages in p's DRAM page group currently contain ML0 pages, DyLeCT demotes one of these ML0 pages to ML1 (i.e., switches its short CTE to long CTE to migrate it to a free DRAM page tracked by the Free List); DyLeCT compares the access counters of these ML0 pages to select the coldest ML0 page to demote.

If the compression using Recency List (see Section II-B) selects an ML0 page as victim, the page is compressed and demoted to ML2 (i.e., switches from short CTE to long CTE).

Figure 12 summarizes promotion and demotion between memory levels as a flowchart.

C. Addressing the Cacheablity Challenge of Short CTEs via a Pre-gathered Table of Short CTEs

To minimize waste in the CTE cache and improve hit rate, DyLeCT gathers copies of short CTEs densely together into a second CTE table that is optimized for short CTEs. Each 64B block in this second table densely packs 64B/2bit=256 short CTEs back-to-back, without wasting any space. As such, each block provides a translation reach of 256*4KB = 1MB, similar to a huge page. Unlike the naive short CTE cache designs in Section IV-A2, which gathers the short CTEs from an unified CTE block into a short CTE cacheline after fetching the CTE block on a CTE miss, DyLeCT proactively gathers/copies the short CTE of a page from the unified CTE table to this second table when promoting the page to ML0. As such, we call this second CTE table the *Pre-gathered Table*.

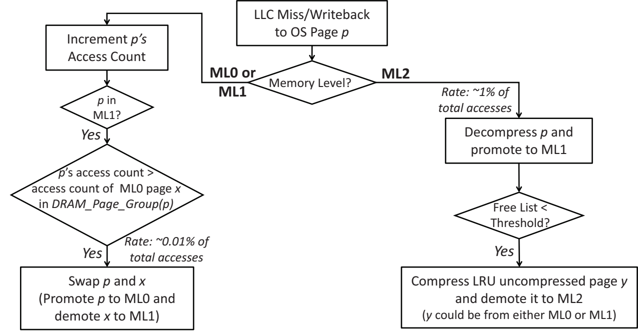


Figure 12: Page management in Three-Level Memory Hierarchy.

Figure 13 shows the internal organization of the Pre-gathered Table. Pre-gathered Table is statically sized to contain a 2-bit entry for every 4KB OS page in the system. For OS pages that use long CTEs (i.e., OS pages in ML1 or ML2), the short CTE in Pre-gathered table records an *INVALID* flag value; the flag value is the maximum encodable number (e.g., '3' for 2-bit short CTEs). As such, 2-bit short CTEs only support three DRAM pages per DRAM page group. DyLeCT updates the short CTE in the Pre-gathered Table whenever it updates the unified CTE table (i.e., when it promotes/demotes a page between memory levels).

To maximize CTE cache hit rate and store as many short CTEs as possible, *DyLeCT features a single CTE cache that stores both pre-gathered and unified blocks*. A single CTE cache inherently allows dynamic partitioning as per workload execution. Unlike the TLB, which is physically split across different dedicated TLBs for 2MB and 4KB PTEs to provide

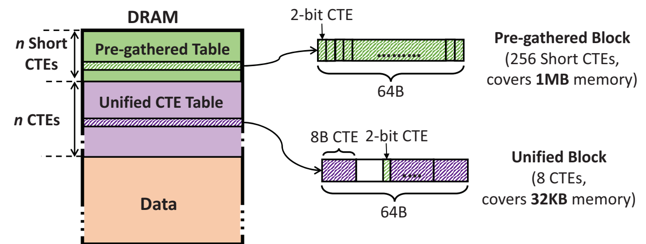


Figure 13: The figure shows Pre-gathered Table and Unified CTE Table in DRAM. Pre-gathered Table stores tightly-packed short CTEs; it is 2b/8B=32x smaller than Unified CTE Table. A 64B block of the Pre-gathered Table enables DyLeCT to fetch up to 256 short CTEs (i.e., a pre-gathered block covers up to 1MB of OS-visible memory). n is the total number of 4KB OS pages in the system. Each 64B Unified Block also contains a dedicated bit per constituent CTE to mark the CTE as long or short.

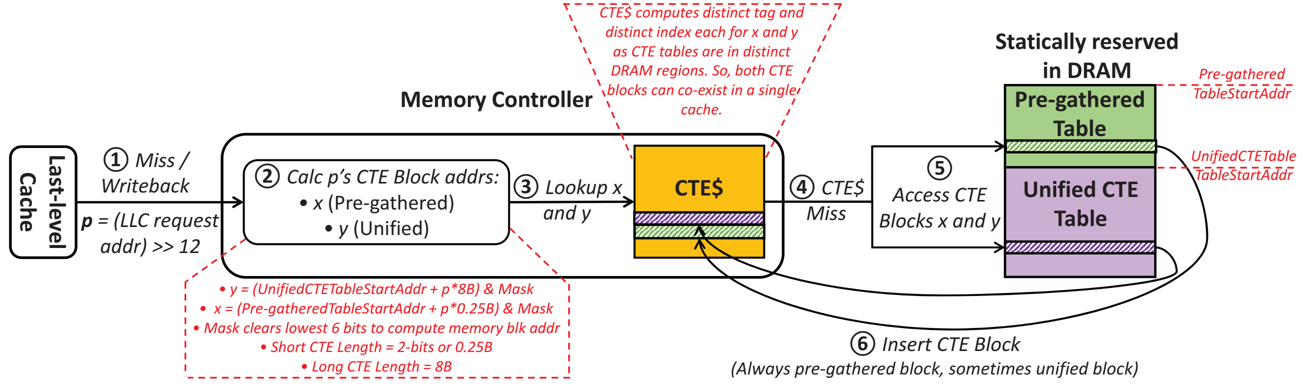


Figure 14: Accessing Pre-gathered and Unified CTE tables during a CTE$ miss.

high bandwidth as TLBs are accessed for every instruction, a unified CTE cache is feasible at the memory controller level, where the access rate is much lower.

As a single CTE cache stores blocks from two distinct CTE tables in DyLeCT, its behavior during hit and miss deviates from prior works, which have only one type of CTE blocks.

1) CTE Cache Hit: We define a CTE cache hit as when a memory access can be fulfilled by a cached CTE block, regardless of whether it is a pre-gathered block or an unified block. For a memory access to page p, DyLeCT first calculates the address of p's pre-gathered block (see ② in Figure 14) and then looks up the CTE cache for the pre-gathered block. If the pre-gathered block hits in the cache, DyLeCT checks whether the short CTE is valid. If it is, DyLeCT uses the hashing function and short CTE to compute the DRAM page address of the requested data. Else, DyLeCT looks for the unified block in the cache; if cache hit, DyLeCT uses the long CTE to access data. Figure 15 summarizes how DyLeCT looks up the CTE cache to obtain short CTEs and long CTEs to serve memory requests.

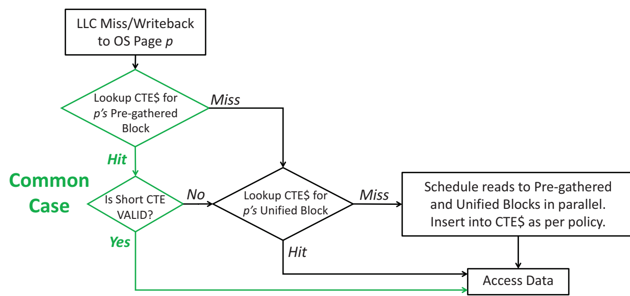


Figure 15: How the memory controller accesses data in memory. The common case (in green) is when short CTE is valid (i.e., OS Page p is in ML0).

2) CTE Cache Miss: We define a CTE cache miss as when the DRAM address for a memory request cannot be determined by the CTE cache. As such, for a request to an ML0 page, a cache miss occurs when both the pre-gathered block and unified block are missing; for a request to an ML1 or ML2 page, however, a cache miss occurs when the unified block misses in the cache.

At the time of each CTE miss, DyLeCT does not know which memory level the requested page belongs to. The naive option is to first assume the page is in ML0 and fetch the pre-gathered block; if the assumption turns out wrong, then sequentially fetch the unified block. However, this can double the CTE access time. As such, DyLeCT fetches both blocks in parallel; this can preserve the same CTE cache miss latency as prior works (see Figure 16). Although this increases bandwidth overhead per CTE cache miss, *the aggregate bandwidth overhead due to CTE cache misses is small (see Figure 23 in Section VI) because DyLeCT significantly reduces overall CTE cache miss rate.*

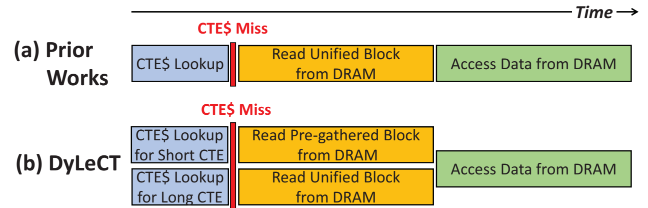


Figure 16: Comparing the timeline of a CTE cache miss in DyLeCT to prior works. DyLeCT fetches both CTE blocks in parallel to avoid increasing the latency of CTE misses. If the memory request is to an ML0 page, as soon as either one of the two CTE block arrives, the ensuing data access can begin; otherwise, the ensuing data access begins only after the unified block arrives.

As DyLeCT always fetches both CTE blocks per CTE miss, it is presented with an option to selectively cache one of the two blocks or to cache both. DyLeCT *always* caches the pre-gathered block, which provides high translation reach; DyLeCT ***also*** caches the unified block ***only*** if the memory request suffering from the CTE miss is to an ML1 or ML2 page.

D. Overheads

Memory: Each short CTE needs a 2 bits per 4KB OS page; therefore, the overhead of Pre-gathered Table is an additional 0.006% of DRAM. The ML1→ML0 promotion policy requires keeping a 5-bit access counter for every 4KB OS page [50]. The overhead of these counters is 0.015% of DRAM.

Logic: Compared to the recent prior work TMCC, DyLeCT is simpler due to only modifying the memory controller, without touching the page walker and caches. However, compared to TMCC, DyLeCT requires adding logic for: (i) DRAM page calculation through a static hash function that uses OS page address and its 2-bit Short CTE, and (ii) ML1→ML0 promotion policy which requires fetching a memory block of access counters for comparison to identify cold victims. These components can be implemented via basic shifters and comparators, similar to a standard cache replacement policy. DyLeCT also adds simple management logic for Pre-gathered Table to update short CTE in Pre-Gathered Table when DyLeCT moves a page between memory levels; this update requires only one extra memory access.

Systems with multiple memory controllers: Like all prior works, DyLeCT is a module within the memory controller (MC). On a system with multiple MCs, each MC has its own DyLeCT module that only compresses the data within its locally-attached DRAM. As such, each MC manages its own hardware data structures (e.g., the CTE tables) and has no need for coherence across MCs. As each DyLeCT module is local to one MC, each page can only be interleaved across the many channels *within one MC*, instead of all channels *across all MCs*. Prior work [27] reports such a slightly restricted memory interleaving has minimal impact on performance.

V. METHODOLOGY

We use Gem5 [4] interfaced with Ramulator [18] as the simulation platform to evaluate DyLeCT's performance. We use DRAMPower [5] for modeling memory subsystem power. We evaluate large and irregular workloads used by recent works on hardware memory compression and address translation [23], [27], [30]. Specifically, we evaluate the same workloads as TMCC - nine benchmarks from GraphBig [24], two benchmarks (*mcf, omnetpp*) from SPEC CPU2017 [42], and one benchmark (*canneal*) from PARSEC 3.0 [3]. For GraphBig and *canneal*, we evaluate multi-threaded execution. For mcf and omnetpp, we run four single-threaded instances for the evaluation.

We run applications under huge pages with Linux's libhugetlbfs library [21] to avoid randomness and reproducibility problems under transparent huge pages. We use Gem5's

CPU	4 cores, 2.8GHz, 4-wide OoO, TLBs: 1024, RoB size: 224
Caches	32KB L1D$, 32KB L1I$, 256KB L2$, 1KB per core walker cache [23], 2MB L3$ per core (8MB total)
Cache Latency	L1$ Hit: 3 clk, L2$ Hit: 14 clk, L3$ Hit: 67 clk (Latencies measured from CPU core; Latencies are accumulative of higher level lookups)
Prefetchers	Next-line with automatic enable/disable: L1$, L2$ Stride: L1$ (degree 2), L2$ (degree 4)
Memory	DDR4-3200, 1 Channel, 8 Ranks, FR-FCFS policy with bank fairness and row buffer hit cap, tCL: 13.75ns, tRCD: 13.75ns, tRP: 13.75ns
CTE Cache	**DyLeCT:** 128KB, 1MB reach per 64B Pre-gathered Block, 32KB reach per 64B Unified Block **TMCC:** 128KB, 32KB reach per 64B CTE block **Hit Latency:** 2 memory clk [6]

Table 3: Simulated microarchitecture parameters.

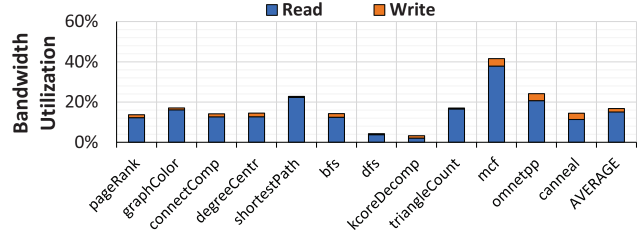


Figure 17: Bandwidth utilization of the evaluated benchmarks during the simulated time window, assuming a conventional system without compression.

KVM mode to fast-forward every benchmark into its region of interest. Next, we fetch all of the benchmark's memory values to place, compress, and pack them into the available DRAM. We then simulate 5 seconds (>20 billion instructions) under Gem5's atomic mode to warm up DyLeCT's memory levels. Finally, we use Gem5's cycle-accurate simulation to warm up prefetchers and branch predictors for 10ms and then use Gem5's cycle-accurate simulation for 40ms to evaluate performance. *We use the total number of committed store instructions per cycle as the metric of performance.* Figure 17 characterizes the memory behavior of the evaluated benchmarks during the simulated time window; the memory bandwidth characterization assumes a conventional system without memory compression.

Modeling the Baseline (TMCC): Since TMCC adaptively compresses memory according to memory demand, we evaluate multiple compression settings; we evaluate the same low and high compression setting as in the TMCC paper. Table 2 describes the two settings in more detail. The low compression setting corresponds to an average compression ratio of 1.3x, while the high compression setting corresponds to an average compression ratio of 2.8x. Note that the TMCC optimization of embedding CTEs within PTBs is not applicable in our evaluation using huge pages.

Modeling DyLeCT: We evaluate 2-bit CTEs, which support three DRAM pages per DRAM page group. We evaluate DyLeCT under the same compression settings as the baseline and use the same compression algorithm and, thus, compression/decompression latencies as the baseline.

Modeling a bigger system without memory compression: We also model a bigger memory system with no compression. This system does not suffer from any compression-related performance overhead because it can fit everything in memory without compression. As such, it incurs no compression-related overheads (i.e., no translation overhead, no decompression latency, no bandwidth overhead).

VI. RESULTS

Figure 18 shows the performance of DyLeCT normalized to TMCC at two different compression settings described in Section V. On average across all benchmarks, the performance benefit of DyLeCT is 10.25%. At low and high compression, the benefits are 11% and 9.5%, respectively. Figure 18 also shows the hypothetical upper-bound performance if CTEs were hypothetically to always hit in the CTE cache. We

see that DyLeCT's performance is close to the upper bound performance at both compression settings.

At low compression, *canneal* gets the highest benefit of 17%. *canneal* has a highly irregular access pattern; in our real-system experiments earlier in Figure 3, we find that *canneal* has the highest TLB misses per LLC miss when the benchmarks use standard 4KB pages. However, *canneal* also suffers the largest drop in performance going from low compression to high compression; its performance benefit drops to 10% at high compression. This is because under high compression, *canneal's* CTE cache hit rate reduces from 93% to 74% (see Figure 19); under high compression, fewer pages can be uncompressed and, therefore, fewer pages are in ML0. For all other benchmarks, however, performance benefit at high compression is very close to benefit at low compression.

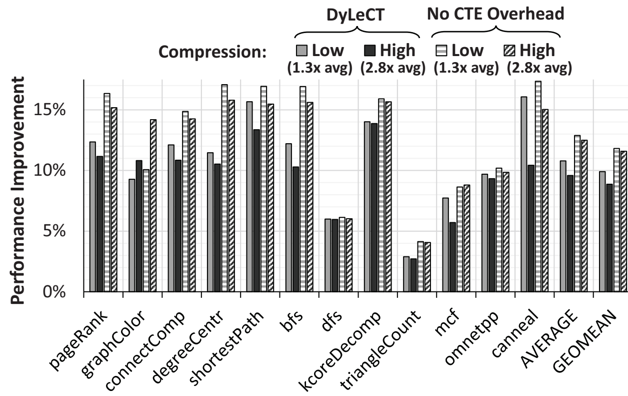


Figure 18: Performance improvement over TMCC.

CTE Cache Hit Rate: The performance improvement comes from improving CTE cache hit rate. Figure 19 shows CTE cache hit rates. At low compression, DyLeCT increases the overall average hit rate from 70% to 96%. At high compression, DyLeCT improves the average CTE cache hit rate from 67% to 91%. At high compression, DyLeCT's hit rate reduces compared to low compression; this is due to relatively fewer pages in ML0 as more pages are compressed into ML2 under high compression (see Figure 20).

Figure 19 also shows the split between pre-gathered block hit and unified block hit for DyLeCT. We note that pre-gathered blocks contribute the vast majority of CTE cache hits in DyLeCT owing to their high 1MB reach. At high compression, the hit rate of pre-gathered blocks is 77% on average. The remaining 14% hits are due to unified blocks.

Latency: DyLeCT's high CTE cache hit rate helps reduce the average L3 miss latency compared to TMCC. Figure 21 shows how much TMCC and DyLeCT increases L3 miss latency compared to a system with no compression. At low and high compression, the average increase under DyLeCT is 2.9ns and 5.8ns. This is substantially lower than TMCC, which increases L3 latency by 9.5ns and 12.8ns at low and high compression, respectively.

Traffic: Figure 22 shows total memory traffic per instruction for DyLeCT normalized to TMCC; it is 93%, on average.

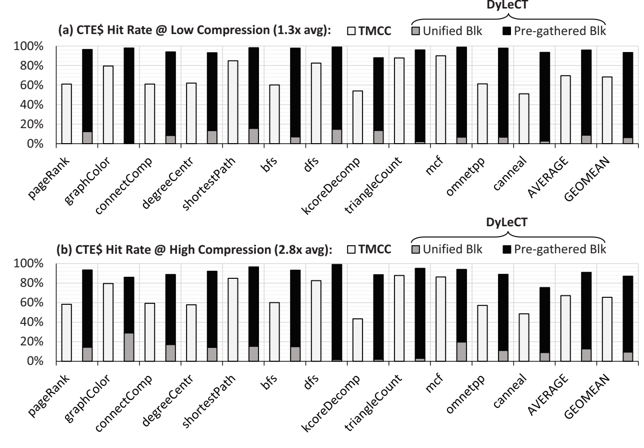


Figure 19: CTE cache hit rate for DyLeCT at (a) low and (b) high compression. Pre-gathered blocks in DyLeCT serve the majority of CTE cache hits. At high compression, CTE cache hit rate is 91% (i.e., the CTE cache serves translations for 91% of memory requests). Pre-gathered blocks serve 77% of memory requests; unified blocks serve the remaining 14%.

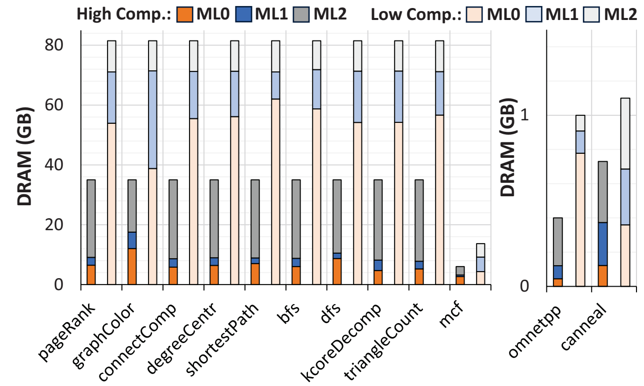


Figure 20: DRAM breakdown of ML0, ML1 and ML2 at high and low compression. Under low compression, the size of ML0 scales up gracefully.

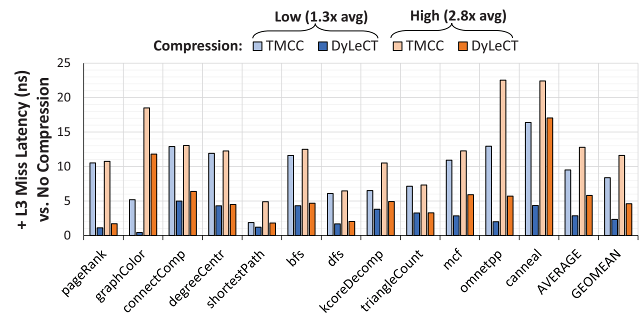


Figure 21: How much DyLeCT and TMCC increases L3 miss latency over a system with No Compression. The latency overhead is shown in nanoseconds.

The total traffic includes all memory accesses – workload's memory accesses, page migration traffic, and memory accesses to CTE blocks due to CTE cache misses.

Even though DyLeCT accesses two CTE blocks per CTE cache miss (i.e., access both the pre-gathered block and unified block), DyLeCT reduces CTE access traffic compared to

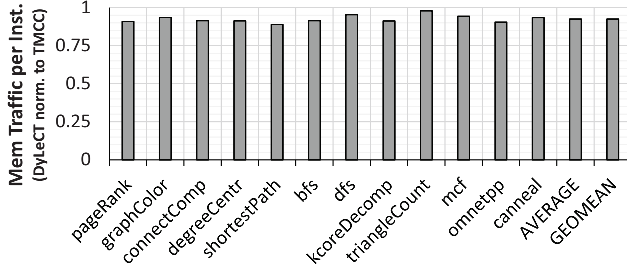


Figure 22: Memory traffic per instruction normalized to TMCC.

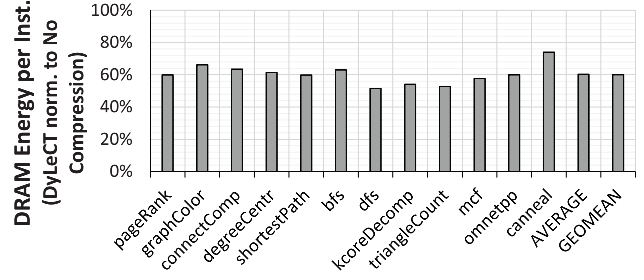


Figure 24: DRAM energy per instruction normalized to No Compression.

TMCC (see Figure 23). This is because DyLeCT significantly reduces CTE miss rate compared to TMCC.

Figure 23 also shows the total memory traffic normalized to TMCC. DyLeCT's total memory traffic is 4.5% higher than TMCC. This increase in total memory traffic is due to increased performance (i.e., more number of committed instructions), which leads to more memory accesses per second.

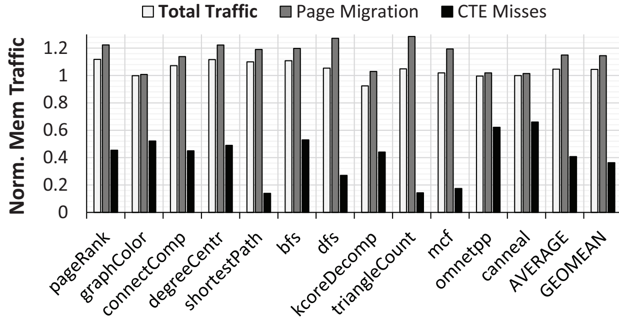


Figure 23: Memory traffic normalized to TMCC. Total memory traffic accounts for all memory accesses (i.e., it includes each workload's memory requests, page migration traffic, and CTE cache misses).

DRAM Energy per instruction: By packing data more densely in memory, hardware memory compression can help reduce the carbon footprint of server memory by reducing how many physical DRAM chips the system requires, which reduces idle (e.g., refresh, standby) memory power; because server memory are typically large, idle memory power tend to dominate overall server memory power.

Figure 24 shows DRAM energy per instruction of DyLeCT normalized to a bigger conventional mmemory system without hardware memory compression. We evaluate the bigger conventional system by using 2x as many DRAM chips as DyLeCT (i.e., evaluate 16 ranks versus 8 ranks). On average across all benchmarks, DRAM energy per instruction is only 60% of a system without memory compression.

Beyond saving on memory energy per instruction, reducing the number of physical DRAM chips can also reduces the embedded carbon footprint of memory (e.g., by 2x when reducing the number of DRAM chips by half).

Sensitivity Analysis - DRAM page group size: We vary the DRAM page group size (i.e., # of pages addressable by a short CTE) to observe the change in the fraction of uncompressed pages in ML0 (see Figure 25). As DRAM page group size increases, the fraction of uncompressed pages in ML0 also increases. Figure 25 also shows that the fraction of uncompressed pages in ML0 does not differ much between DRAM page group size of 3 (i.e., 2-bit short CTEs) and 7 (i.e., 3-bit short CTEs). Without an increase in fraction of uncompressed pages in ML0, using 3-bit short CTEs would reduce CTE cache's translation reach and slightly degrade performance. Therefore, using 2-bit short CTEs is the sweet spot for DyLeCT. With DRAM page group size of 3, 66% of uncompressed pages are in ML0.

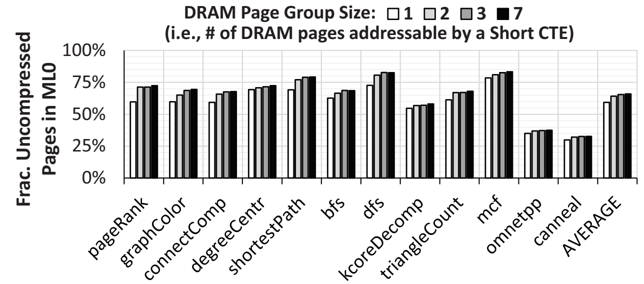


Figure 25: Fraction of Uncompressed Pages in ML0 when varying DRAM page group size (i.e., # of pages addressable by a Short CTE). Fraction of Uncompressed Pages in ML0 = # ML0 Pages/Total # Uncompressed Pages). The results in the figure correspond to high memory compression.

VII. Discussion and Other Related Work

DyLeCT is fully compatible with virtualization similar to all prior works on hardware memory compression. Virtualization adds a guest pseudo-physical address *before* the host physical address whereas hardware memory compression adds a translation (i.e., CTE) *after* the host physical address. Moreover, as CTEs form a layer beyond the host physical address, all hardware memory compression works including DyLeCT function correctly for all OS page sizes (e.g., 1GB). Therefore, DyLeCT is agnostic of the page size(s) the OS uses.

Near-memory Address Translation [36] and Mosaic Pages [10] propose using shorter translations to speed up address translation in the context of ***conventional*** OS-managed virtual-to-physical address translation. Unlike DyLeCT, which dynamically switches the MC-managed translations (i.e., CTEs)

of individual pages between long and short translations, these works propose *entirely* replacing the conventional long translations with short translations. Unlike the new address translation layer (i.e., CTE layer) required by hardware memory compression, the conventional virtual-to-physical address translation layer is managed by the OS; this makes dynamically updating address translation several orders of magnitude more costly than switching the length of translations under DyLeCT. To use short translations on every page while keeping the memory loss small, these prior works also use many times longer short translations than DyLeCT (e.g., 7 bits per translation, instead of 2 bits). Lastly, using short translations on every page also simply won't work for hardware memory compression.

VIII. Conclusion

This paper proposes Dynamic Length Compressed-Memory Translations (DyLeCT) to achieve huge-page-like translation performance in the new layer of address translation required by hardware memory compression. DyLeCT dynamically switches between short translation and long translation for each page individually. DyLeCT uses cache-friendly short translations on the hottest pages to improve overall translation performance and uses long translations on the colder pages to preserve high effective memory capacity. For large and irregular applications that use huge pages, DyLeCT significantly increases CTE cache hit rate and, thus, provides 10.25% performance improvement over the prior art, while maintaining the same compression ratio. DyLeCT is also a simple design that modifies only the memory controller.

Acknowledgment

We thank the National Science Foundation (NSF) for generously supporting this work through grants 1942590, 1919113, and 2312785. We also thank Advanced Research Computing (ARC) at Virginia Tech for providing computational resources.

References

[1] S. Ainsworth and T. M. Jones, "Compendia: Reducing virtual-memory costs via selective densification," in *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 52–65. [Online]. Available: https://doi.org/10.1145/3459898.3463902

[2] J.-L. Baer, "Sectored (or subblock) caches." [Online]. Available: https://courses.cs.washington.edu/courses/csep548/00sp/lectures/class5/sld058.htm

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 72–81. [Online]. Available: https://doi.org/10.1145/1454115.1454128

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[5] K. Chandrasekar, C. Weis, Y. Li, S. Goossens, M. Jung, O. Naji, B. Akesson, N. Wehn, and K. Goossens, "Drampower: Open-source dram power & energy estimation tool." [Online]. Available: http://www.drampower.info

[6] E. Choukse, M. Erez, and A. R. Alameldeen, "Compresso: Pragmatic main memory compression," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 546–558. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00051

[7] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ser. ISCA '05. USA: IEEE Computer Society, 2005, p. 74–85. [Online]. Available: https://doi.org/10.1109/ISCA.2005.6

[8] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: Association for Computing Machinery, 2012, p. 37–48. [Online]. Available: https://doi.org/10.1145/2150976.2150982

[9] M. Gorman and P. Healy, "Performance characteristics of explicit superpage support," in *Proceedings of the 2010 International Conference on Computer Architecture*, ser. ISCA'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 293–310. [Online]. Available: https://doi.org/10.1007/978-3-642-24322-6_24

[10] K. Gosakan, J. Han, W. Kuszmaul, I. N. Mubarek, N. Mukherjee, K. Sriram, G. Tagliavini, E. West, M. A. Bender, A. Bhattacharjee, A. Conway, M. Farach-Colton, J. Gandhi, R. Johnson, S. Kannan, and D. E. Porter, "Mosaic pages: Big tlb reach with small pages," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 433–448. [Online]. Available: https://doi.org/10.1145/3582016.3582021

[11] H. Hadian, M. Farrokh, M. Sharifi, and A. Jafari, "An elastic and traffic-aware scheduler for distributed data stream processing in heterogeneous clusters," *The Journal of Supercomputing*, vol. 79, no. 1, pp. 461–498, Jan 2023. [Online]. Available: https://doi.org/10.1007/s11227-022-04669-z

[12] H. Hadian and M. Sharifi, "Gt-scheduler: a hybrid graph-partitioning and tabu-search based task scheduler for distributed data stream processing systems," *Cluster Computing*, Feb 2024. [Online]. Available: https://doi.org/10.1007/s10586-023-04260-y

[13] S. Hong, P. J. Nair, B. Abali, A. Buyuktosunoglu, K.-H. Kim, and M. Healy, "Attaché: Towards ideal memory compression by mitigating metadata bandwidth overheads," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 326–338.

[14] A. Hunter, C. Kennelly, P. Turner, D. Gove, T. Moseley, and P. Ranganathan, "Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 257–273. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/hunter

[15] Intel, "Intel xeon w-3175x processor," Last accessed on Jul 31, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/189452/intel-xeon-w3175x-processor-38-5m-cache-3-10-ghz/specifications.html

[16] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-plane compression: Transforming data for better compression in many-core architectures," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 329–340.

[17] S. Kim, S. Lee, T. Kim, and J. Huh, "Transparent dual memory compression architecture," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 206–218.

[18] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 45–49, Jan. 2016. [Online]. Available: https://doi.org/10.1109/LCA.2015.2414456

[19] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Ingens: Huge page support for the os and hypervisor," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 1, p. 83–93, sep 2017. [Online]. Available: https://doi.org/10.1145/3139645.3139659

[20] T. Lee, S. K. Monga, C. Min, and Y. I. Eom, "Memtis: Efficient memory tiering with dynamic page classification and page size determination," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association

for Computing Machinery, 2023, p. 17–34. [Online]. Available: https://doi.org/10.1145/3600006.3613167
[21] libhugetlbfs, "libhugetlbfs." [Online]. Available: https://github.com/libhugetlbfs/libhugetlbfs
[22] Linux Kernel Docs, "Transparent hugepage support," Last accessed on Jul 31, 2023. [Online]. Available: https://docs.kernel.org/admin-guide/mm/transhuge.html
[23] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched address translation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 1023–1036. [Online]. Available: https://doi.org/10.1145/3352460.3358294
[24] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "Graphbig: Understanding graph computing in the context of industrial solutions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807626
[25] A. Panwar, S. Bansal, and K. Gopinath, "Hawkeye: Efficient fine-grained os support for huge pages," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 347–360. [Online]. Available: https://doi.org/10.1145/3297858.3304064
[26] A. Panwar, A. Prasad, and K. Gopinath, "Making huge pages actually useful," *SIGPLAN Not.*, vol. 53, no. 2, p. 679–692, mar 2018. [Online]. Available: https://doi.org/10.1145/3296957.3173203
[27] G. Panwar, M. Laghari, D. Bears, Y. Liu, C. Jearls, E. Choukse, K. W. Cameron, A. R. Butt, and X. Jian, "Translation-optimized memory compression for capacity," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 992–1011.
[28] G. Panwar, D. Zhang, Y. Pang, M. Dahshan, N. DeBardeleben, B. Ravindran, and X. Jian, "Quantifying memory underutilization in hpc systems and using it to improve performance via architecture support," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 821–835. [Online]. Available: https://doi.org/10.1145/3352460.3358267
[29] C. H. Park, S. Cha, B. Kim, Y. Kwon, D. Black-Schaffer, and J. Huh, "Perforated page: Supporting fragmented memory allocation for large pages," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 913–925.
[30] C. H. Park, I. Vougioukas, A. Sandberg, and D. Black-Schaffer, *Every Walk's a Hit: Making Page Walks Single-Access Cache Hits*. New York, NY, USA: Association for Computing Machinery, 2022, p. 128–141. [Online]. Available: https://doi.org/10.1145/3503222.3507718
[31] S. Park, I. Kang, Y. Moon, J. H. Ahn, and G. E. Suh, "Bcd deduplication: effective memory compression using partial cache-line deduplication," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 52–64. [Online]. Available: https://doi.org/10.1145/3445814.3446722
[32] A. Patil, V. Nagarajan, R. Balasubramonian, and N. Oswald, "Dvé: improving dram reliability and performance on-demand via coherent replication," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. IEEE Press, 2021, p. 526–539. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00048
[33] G. Pekhimnko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 172–184.
[34] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *Proceedings of USENIX Security'16*, 2016.
[35] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach tlbs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 258–269.
[36] J. Picorel, D. Jevdjic, and B. Falsafi, "Near-memory address translation," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2017, pp. 303–317. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/PACT.2017.56
[37] C. Qian, L. Huang, Q. Yu, Z. Wang, and B. Childers, "Cmh: Compression management for improving capacity in the hybrid memory cube," in *Proceedings of the 15th ACM International Conference on Computing Frontiers*, ser. CF '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 121–128. [Online]. Available: https://doi.org/10.1145/3203217.3203235
[38] V. S. S. Ram, A. Panwar, and A. Basu, "Trident: Harnessing architectural resources for all page sizes in x86 processors," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1106–1120. [Online]. Available: https://doi.org/10.1145/3466752.3480062
[39] RedHat, "How to use, monitor, and disable transparent hugepages in red hat enterprise linux 6 and 7?" Last accessed on Jul 31, 2023. [Online]. Available: https://access.redhat.com/solutions/46111
[40] J. Rothman and A. Smith, "Sector cache design and performance," in *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No.PR00728)*, 2000, pp. 124–133.
[41] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, *Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1093–1108. [Online]. Available: https://doi.org/10.1145/3373376.3378493
[42] Standard Performance Evaluation Corporation, "Spec cpu2017," https://www.spec.org/cpu2017/.
[43] R. Tremaine, T. Smith, M. Wazlowski, D. Har, K.-K. Mak, and S. Arramreddy, "Pinnacle: Ibm mxt in a memory controller chip," *IEEE Micro*, vol. 21, no. 2, pp. 56–68, 2001.
[44] VMWare, "Performance best practices for vmware cloud on aws," 2021. [Online]. Available: https://docs.vmware.com/en/VMware-Cloud-on-AWS/services/vmc-aws-performance.pdf
[45] J. Wang and M. Balazinska, "Elastic memory management for cloud data analytics," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '17. USA: USENIX Association, 2017, p. 745–758.
[46] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang, and D. Skarlatos, "Tmo: Transparent memory offloading in datacenters," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 609–621. [Online]. Available: https://doi.org/10.1145/3503222.3507731
[47] Wikichip, "Skylake (server) - microarchitectures - intel," Last accessed on Jul 31, 2023. [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)
[48] V. Young, S. Kariyappa, and M. K. Qureshi, "CRAM: efficient hardware-based memory compression for bandwidth enhancement," *CoRR*, vol. abs/1807.07685, 2018. [Online]. Available: http://arxiv.org/abs/1807.07685
[49] V. Young, S. Kariyappa, and M. K. Qureshi, "Enabling transparent memory-compression for commodity memory systems," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 570–581.
[50] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-efficient dram caching via software/hardware cooperation," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–14. [Online]. Available: https://doi.org/10.1145/3123939.3124555
[51] D. Zhang, G. Panwar, J. B. Kotra, N. DeBardeleben, S. Blanchard, and X. Jian, "Quantifying server memory frequency margin and using it to improve performance in hpc systems," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. IEEE Press, 2021, p. 748–761. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00064
[52] J. Zhao, S. Li, J. Chang, J. L. Byrne, L. L. Ramirez, K. Lim, Y. Xie, and P. Faraboschi, "Buri: Scaling big-memory computing with hardware-based memory expansion," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 3, oct 2015. [Online]. Available: https://doi.org/10.1145/2808233
[53] K. Zhao, K. Xue, Z. Wang, D. Schatzberg, L. Yang, A. Manousis, J. Weiner, R. Van Riel, B. Sharma, C. Tang, and D. Skarlatos,

"Contiguitas: The pursuit of physical memory contiguity in datacenters," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589079

[54] W. Zhu, A. L. Cox, and S. Rixner, "A comprehensive analysis of superpage management mechanisms and policies," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 829–842. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/zhu-weixi